# Feuille de TP Python 01

Introduction à Python en BCPST

## 1 Python en BCPST kezako ?

Python est un langage de programmation généraliste et universel. Il a notamment été adopté par le MIT pour ses cours d'introduction à l'informatique. Il est disponible sur tous les types de machines contemporaines.

Connaître le b-a-ba de Python, c'est donc mettre un pied dans le monde de l'informatique au sens le plus large possible.

Connaître le b-a-ba de Python, et ce sera notre modeste but cette année, est aussi un moyen efficace de tester, expérimenter, simuler et illustrer les notions au programme de l'année. Nous allons donc explorer le versant scientifique de Python.

L'objectif de cette série de TD/TP est d'une part de vous permettre de développer un petit projet, dont le thème sera lié à celui de votre TIPE, que vous devrez exposer à l'oral et d'autre part, d'acquérir le minimum d'aisance pour être en mesure d'aborder les questions de programmation et d'interprétations graphiques lors des épreuves d'écrit et d'oral.

Il n'est pas possible d'être exhaustif et, pour les compléments d'information, quelques recherches sur Internet ou dans l'aide en ligne de Python deviennent très vite nécessaires.

## 2 Tous premiers pas en Python

Il faudra que vous ayiez très rapidement sur vos machines personnelles l'environnement de développement Pyzo/Python3.3 disponible à l'adresse

`http://www.pyzo.org/downloads.html`

En attendant d'avoir une installation fonctionnelle, on peut aller sur PythonTutor,

`http://pythontutor.com/visualize.html#mode=edit`

qui va nous permettre d'illustrer notre propos préliminaire.

**Il faut choisir d'utiliser Python3.3, sinon le code ne se comportera pas toujours comme prévu.**

Vous pouvez copier/coller les bouts de code du texte dans PythonTutor. Il faut faire attention à respecter les indentations (le nombre d'espace au début de chaque ligne) car celles-ci ont une signification importante dans le langage.

PythonTutor permet, via une exécution pas à pas, de comprendre comment et surtout dans quel ordre les instructions s'exécutent. Il montre comment se créent et disparaissent des « objets »(nombres, fonctions, tableaux...) dans la mémoire de l'ordinateur. Son utilisation est limitée à de petits programmes mais est très instructive.

Python est un langage *interprété* qui peut s'utiliser par l'intermédiaire d'une console dans laquelle on entre les commandes ou par l'exécution d'un fichier texte–un *script* ou un *programme* –qui contient les commandes Python.

### 2.1 Objets et étiquettes

Le premier point à comprendre est que Python traite avec des *objets*. C'est la façon dont Python stocke les données, les organise et permet leur utilisation.

Ces objets sont, pratiquement parlant, des zones de la mémoire identifiées, une telle zone contenant le code décrivant un objet. N'oubliez jamais qu'une zone mémoire contient une suite de bits, chacun d'entre eux pouvant valoir 0 ou 1. Un objet Python (et plus généralement, tout ce que vous rencontrez dans votre environnement informatisé quotidien) est codé par une suite suffisamment longue de 0 et de 1.

Chaque objet possède une *identité*, un *type* et une *valeur*. Un objet est référencé dans le script par une *étiquette* aussi appelé un *identificateur*

L'utilisation de ces étiquettes est similaire à celle des variables d'un autre langage de programmation ou d'un fragment de discours mathématique.

Le *type* d'un objet Python est similaire au « type »d'un objet mathématique : nombres, fonctions, ensembles, matrices,...

On créé un objet en écrivant une *expression*, on lui affecte–= est l'opérateur d'*affectation* – une étiquette, on l'utilise en le référençant par cette étiquette et, quand on n'a plus besoin de lui, soit on le détruit volontairement (instruction `del`), soit il se fait détruire par l'interpréteur.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
a=1.2
print(type(a))
print(id(a))
b=a
print(id(b))
a=1.3
print(id(a),id(b))
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Que fait-on, que voit-on ?

1. Première étape : On a créé l'objet (un nombre réel) `1.2` de *type* `float`, que l'on a étiqueté `a`. On voit cet objet apparaître dans la mémoire principale (Global Frame) de l'interpréteur sur la droite.
2. Deuxième étape : on demande à Python d'afficher (`print`), le type (`type()`) de l'objet référencé par l'étiquette `a`
3. Troisième étape : on demande à Python d'afficher (`print`), l'identité (`id()`) de l'objet référencé par l'étiquette `a`. C'est un code interne à l'interpréteur dont nous ne nous servirons pas
4. Quatrième étape : on introduit une étiquette `b` et l'on déclare que l'objet référencé par `b` est le même que celui référencé par `a`.
5. Cinquième étape : on a créé l'objet de type `float 1.3` et on le référence à l'aide de l'étiquette `a`.
6. On demande d'afficher les identités de `a` et `b`. On constate que celles-ci sont différentes.

On peut faire des opérations sur des objets pour créer de nouveaux objets que l'on peut étiqueter. Ici, on va additionner des nombres réels. On peut aussi faire des comparaisons et, suivant le résultat vrai ou faux de cette comparaison, effectuer une ou des opérations.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
a=1.2
b=1.3
c=a+b
print(id(a),id(b),id(c))
a=a+b
print(id(a),id(b),id(c))
if(a==c):
    print('a vaut c')
if(a!=c):
    print('a est different de c')
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Que se passe-t-il ?

1. l'objet `1.2` est créé, l'étiquette `a` le référence
2. l'objet `1.3` est créé, l'étiquette `b` le référence
3. On crée l'objet « somme des objets référencés par `a` et `b` ». Celui-ci est référencé par l'étiquette `c`
4. On affiche les identités de ces différents objets.
5. On recrée l'objet « somme des objets objets référencés par `a` et `b` ». Celui-ci est référencé par l'étiquette `a`.
6. On affiche les identités de ces différents objets. `b`. L'étiquette `a` vient de changer d'objet référencé. Les objets référencés par `a` et `c` ont même valeur, ils n'ont pourtant pas forcément même identité.
7. Notre première instruction de contrôle `if` : On teste l'égalité de `a` et `c` : ici on teste s'ils ont même valeur, ce qui est caractéristique des étiquettes référençant des types numériques. On verra sur d'autres exemples que pour les autres types, la comparaison est une comparaison d'identité.

## 2.2 Les types d'objets fondamentaux

En Python, **tout** est objet. Il y a cependant une hiérarchie des objets relativement à leur complexité. On va en dresser une liste pratique non exhaustive. Pour la liste complète (difficilement compréhensible pour un novice) , on peut se référer à

`https://docs.python.org/3.3/reference/datamodel.html`

Les objets les plus simples sont les objets (de type) numériques simples (booléens, entiers, réels, complexes), viennent ensuite les objets en *suite finie* (chaînes de caractères, *t-uples*, listes) et les objets dictionnaires (hors-programme). Les fonctions et les fichiers sont aussi des objets.

Chaque (type d')objet possède ses propriétés propres, que ce soit au niveau de la récupération de la valeur de l'objet ou des opérations/fonctions le concernant.

### 2.2.1 Les booléens

Il n'y a que deux objets booléens `True` et `False`, représentant les valeurs de vérité vrai (1) et faux (0).

Un booléen est typiquement la valeur de vérité d'une expression de comparaison. La valeur de vérité de « 2 est égal à 0 »est *faux*, celle de « 2 est égal à 0 **ou** 2 est pair »est *vrai*. Un booléen est l'argument naturel d'une instruction de contrôle de type `if` ou `while`.

Les opérations de la logique usuelle s'appliquent aux booléens pour donner comme résultat un booléen.

---

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

---

```
a=True
b=False
print(type(a),id(a),type(b),id(b))
a=(2==0)
b=(2>0)
c=(2<0)
print(a,b,c)
print(a and b, a or c, not(a and b) or c)
```

---

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

---

Que voit-on ?

1. l'objet `True` est référencé par l'étiquette `a`,
2. l'objet `False` est référencé par l'étiquette `b`,
3. On écrit le type et l'identité de ces objets.
4. l'étiquette `a` référence l'objet booléen résultat du test d'égalité (==) de deux entiers
5. l'étiquette `b` référence l'objet booléen résultat du test de comparaison (>) de deux entiers
6. l'étiquette `c` référence l'objet booléen résultat du test de comparaison (<) de deux entiers
7. on imprime les valeurs de `a`,`b` et `c`,
8. on imprime les résultats des opérations `a` *et* `b`, `a` *ou* `b`, *non*(`a` *et* `b`) *ou* `c`,

### 2.2.2 Les entiers

Un objet de type `int`, représentant un nombre entier se définit soit en le déclarant explicitement sous forme décimale `a=2` soit comme le résultat d'opérations sur des objets menant à un entier p.ex. `a=(b+2)*3` pourvu que `b` soit un entier.

Les opérateurs arithmétiques autorisés sont

1. `+`, la somme, `-`, la différence, `*`, le produit,
2. `/`, la division, ce résultat peut être de type réel si la division ne « tombe pas juste ».
3. `%`, `//`, respectivement le reste et le quotient dans la division euclidienne,
4. `**`, l'exponentiation : `a**b` calcule $a^b$

Les expressions peuvent être parenthésées avec les règles usuelles de priorités.

On peut comparer deux expressions entières, le résultat étant alors un booléen Les opérateurs de comparaison (ils ne sont pas forcément spécifiques aux entiers) sont

1. l'égalité ==
2. la différence `!=`
3. les deux inégalités strictes > et <
4. les deux inégalités larges >= et <=

---

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

---

```
a=2
b=-3
print(type(a),type(b))
print(a+b,a-b,a*b,a/b,a%b)
print(a+b>=a*b)
print(a/b*b,a/b*b+a%b)
print(a//b*b,a//b*b+a%b)
print(b**a,a**b)
```

---

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

---

Expliquer les résultats des deux dernières lignes !

### 2.2.3 Réels et complexes

Un objet de type `float` représentant un nombre réel se définit soit en le déclarant explicitement sous forme décimale `a=2.0`, `pi=3.141592` soit comme le résultat d'opérations sur des objets menant à un nombre réel, p.ex. `c=(b+2)*3+a`.

Les opérateurs arithmétiques et de comparaison sont les mêmes que pour les nombres entiers à l'exception notable de `//` et `%`

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
a=2.0
b=-3.0
print(type(a),type(b))
print(a+b,a-b,a*b,a/b,a%b)
print(a+b>=a*b)
print(a/b*b,a/b*b+a%b)
print(b**a,a**b)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Expliquer les résultats des deux dernières lignes et comparer à l'exemple précédent.

On peut mélanger des opérations portant sur les entiers et les réels, un entier sera alors transformé naturellement en réel. SI par hasard vous utilisez Python2.7 les divisions entières ne se comporte pas de la même façon, ce qui peut être source d'erreur.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
a=2.0
b=-3
print(type(a),type(b))
print(a+b,a-b,a*b,a/b,a%b)
print(a+b>=a*b)
print(a/b*b,a/b*b+a%b)
print(b**a,a**b)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Comparer à l'exemple précédent.

Python comprend les nombres complexes. Le nombre $i$ s'écrit `1j`. Le nombre $3+2i$ s'écrit `3+2j`. `j` doit toujours être précédé d'un nombre entier ou réel.

Les opérateurs arithmétiques sont les mêmes que pour les nombres réels. Les opérateurs de comparaison autres que l'égalité ou la différence n'ont pas de sens (pourquoi ?).

On peut récupérer partie réelle et partie imaginaire d'un objet de type nombre complexe en utilisant la notation d' *attributs* associés à un objet. Un nombre complexe n'est que le couple de ses parties réelle et imaginaire. Si $z$ est un objet de type `complex`, que celui-ci soit référencé par une étiquette, soit le résultat d'une expression, etc...alors `z.real` et `z.imag`, parties réelle et imaginaire de `z` sont des objets de type `float`.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
a=2.0+1j
b=-3+0j
c=-3
print(type(a),type(b),type(c))
print(a+b,a-b,a*b,a/b)
print((a**2).real,(a**2).imag, a**2)
c=a*b-b*a
print(type(c),c)
print(a+b>=a*b)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Au cas où l'on doit faire des calculs plus avancés sur les nombres complexes, il faut utiliser une *librairie* ou un *module* externe. Ceci ne fonctionne pas toujours très bien dans PythonTutor et nous reviendrons sur ce point plus tard, en utilisant Pyzo.

### 2.2.4 Chaînes de caractères

En calcul scientifique, on est peu amenés à effectuer des opérations sur des chaînes de caractères, hormis pour écrire les résultats de nos calculs sous forme lisible, et, peut-être pour certains codages (penser à une chaîne ADN).

Une chaîne de caractère (de type `str`) est une suite finie et ordonnée de caractères, par exemple l'instruction `s='abcdef'` donne l'étiquette `s` au mot formé successivement des lettres `a`,`b`,..,`f`. On utilise des guillemets simples (`'`) ou doubles (`"`) pour encadrer la suite de lettres. Une chaîne peut contenir tous les caractères apparaissant au clavier et d'autres, plus exotiques, comme les caractères de contrôle. Le caractère d'échappement \ joue un rôle particulier dans le codage des caractères exotiques.

Voici quelques commandes pour manipuler les chaînes de caractères. On suppose que `s`,`t`,... sont des objets de type `'str'`

1. `len(s)` : la longueur de `s`, c'est un entier.
2. `s+t` : la chaîne issue de la concaténation de `s` et `t`
3. `s[i]` : le caractère d'indice `i` de la chaîne `s`. Les indices vont de 0 à `len(s)-1`
4. `s[-1]` : le dernier caractère de la chaîne `s`. On peut lire la chaîne à l'envers avec des indices négatifs.
5. `s[i:j]` : la sous-chaîne comprise entre les indices `i` et `j`
6. `x in s` : retourne un booléen, vrai si et seulement si `x` est une sous-chaîne de `s`
7. `x not in s` : retourne un booléen, vrai si et seulement i `x` n'est pas une sous-chaîne de `s`
8. ...

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
s="abcdef"
t="wxyz"
l=len(s)
m=len(t)
print(type(s),type(t),type(l),type(m),l,m)
print(s+t,type(s+t),len(s+t))
u=s+t
v=u[2:4]
print(u,v)
print('a' in u,'a' in v)
print(min('bcade'))
print(max('acebd'))
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

### 2.2.5 t-uples

Les chaînes que nous venons de voir sont des cas particulier d'un type d'objet « conteneur »plus général : le *t-uple*. Un t-uple permet de grouper un nombre fini d'objets de types hétérogènes en une suite finie et ordonnée.

La syntaxe est d'écrire la liste des objets en commençant par une parenthèse ouvrante, de séparer les objets par des virgules et de finir la liste par une parenthèse fermante.

Les commandes agissant sur une telle suite sont très similaires à celles agissant sur les chaînes de caractères.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
a="abcdef"
pi=3.14
ln2=0.69
tuple_nimp=(a,ln2,pi,True,(3>=2) and (-1>0))
print(type(tuple_nimp))
print(len(tuple_nimp))
tuple_nimp2=(1+2j,"ahah")
print(len(tuple_nimp2))
tuple_nimp3=(tuple_nimp+tuple_nimp2)[2:6]
print(id(tuple_nimp3))
#tuple_nimp3[1]="hihi"
tuple_nimp3=(tuple_nimp3[0],)+("hihi",)+tuple_nimp3[2:]
print(id(tuple_nimp3))
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Il faut absolument remarquer que la ligne avec # est illégale (enlevez le # et faites tourner le script) : **on ne peut pas changer l'identité d'un élément particulier d'un t-uple**. On pourra par contre, construire un nouveau t-uple à partir d'un t-uple existant comme dans l'avant-dernière ligne. La syntaxe `(objet,)` permet de fabriquer un t-uple ne contenant qu'un élément.

### 2.2.6 Listes

Les t-uples, les chaînes de caractères, les nombres sont des objets *immuables*. On ne peut changer leur valeur (la valeur d'un t-uple est la liste des identités des composants).

La dernière structure importante pour cette introduction est une structure « mutable », la structure de *liste*. Disons grossièrement qu'il s'agit de la variante du type t-uple qui autorise la modification des composants.

La syntaxe est d'écrire la liste des objets en commençant par un crochet ouvrant, de séparer les objets par des virgules et de finir la liste par un crochet fermant.

Les commandes agissant sur une telle suite sont très similaires à celles agissant sur les t-uple, auxquelles on ajoute les commandes permettant la modification des composants.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
a="abcdef"
pi=3.14
ln2=0.69
liste_nimp=[a,ln2,pi,True,(3>=2) and (-1>0)]
print(type(liste_nimp))
print(len(liste_nimp))
liste_nimp2=[1+2j,"ahah"]
print(len(liste_nimp2))
liste_nimp3=(liste_nimp+liste_nimp2)[2:6]
print(id(liste_nimp3))
liste_nimp3[1]="hihi"
print(id(liste_nimp3))
liste_nimp.extend(liste_nimp2)
liste_nimp4=liste_nimp[2:6]
print(id(liste_nimp4))
liste_nimp4[3]=2.71j
print(liste_nimp3[3])
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Remarquer la syntaxe de *méthode* associée à un objet utilisée dans la ligne `liste_nimp.extend(liste_nimp2)`. Cette syntaxe est du type `objet.methode(argument)`. Elle permet d'appliquer une fonction à un objet. En quelque sorte, cette syntaxe permet à la fonction de faire partie intégrante de la spécification du type d'objet. Que fait la commande ? Elle dit à liste `liste_nimp` de prendre la liste passée en argument et de se l'ajouter (à elle-même !) en fin de liste.

On peut bien évidemment faire des listes de listes (on peut faire des listes de n'importe quoi !), des listes de t-uples, des t-uples de listes ; Ce peut être utile pour coder des vecteurs, des matrices (une liste de vecteurs), des graphes, etc...

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
liste_disparate=[
        [0,1,2,3],
        ["truc","bidule","chouette"],
        ["truc", 0, 1j]
       ]
matrice=[
          [1,2,3,1,7],
          [3,4,5,3,-2],
          [-1,2,3,4,5],
          [-1,-4,7,8,9]
         ]
print(type(liste_disparate),type(matrice))
print(matrice)
print(matrice[1][2])
print(matrice[2][2:3])
print(matrice[1:3])
print(matrice[1:3][1])
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Expliquer le résultat de la dernière ligne, qui n'est pas forcément celui escompté !

### 2.2.7 Des types d'objets, il y en a d'autres !

Nous avons vu quelques types fondamentaux qui vont nous permettre de débuter le travail. Il ne faut cependant pas perdre de vue qu'il y a de nombreux autres types d'objets : nous aurons affaire en cours de travail aux types dictionnaire, fichier et fonction. Ce dernier type est particulier, il permet par exemple de passer une fonction en paramètre d'une autre, ce qui est très intéressant en calcul numérique.

D'autres types d'objets sont définis dans les modules–nous nous servirons notamment d'un type du module `numpy` qui permet de faire du calcul matriciel. Nous limiterons drastiquement la variété d'objets utilisés ainsi que le nombre de modules.

## 2.3 Blocs de programme et instructions de contrôle

A une exception près (le programme d'introduction contenant des `if`), les programmes que nous avons rencontrés jusqu'à présent ont une exécution « linéaire » : ils sont composés d'une suite d'instructions que l'on exécute l'une après l'autre : dans

PythonTutor, à chaque clic sur Forward>, la flèche verte descend d'une ligne.

Un programme informatique en général n'a pas ce comportement linéaire. On doit avoir la possibilité de sauter directement d'une ligne d'instruction à une autre qui peut être située en amont ou en aval dans le texte. Ce saut peut être conditionné (ou pas) *i.e.* ne s'effectuer que si une certaine condition est vraie.

En Python, ceci est implémenté via trois instructions : `if`, `while` et `for`.

L'instruction `if` permet de sauter « en avant »dans le déroulement du programme. Les deux autres instructions permettent de sauter « en arrière ».

### 2.3.1 Exécution conditionnelle : `if`

L'instruction `if`, dans sa version la plus simple, permet d'exécuter un ensemble d'instructions pourvu qu'un certain objet booléen `p` soit à la valeur `True`. Son insertion dans le programme suit le schéma suivant

```
amont
if p :
    bloc
aval
```

Le déroulement du programme est le suivant : la suite d'instructions `amont` est exécutée jusqu'à ce que le curseur arrive à `if p :`. `p` est un objet (peut-être issu du calcul d'une expression) de type booléen qui est évalué. Deux possibilités s'offrent alors

- Si `p` vaut `True`, la suite d'instructions `bloc` puis la suite d'instructions `aval` sont exécutées.
- Si `p` vaut `False`, la suite d'instructions `bloc` n'est pas exécutée et on saute directement à l'exécution de la suite d'instructions `aval`.

La suite d'instructions `bloc` est appelée le *bloc de programme* associé à l'instruction conditionnelle `if p :`

**Un bloc de programme Python associé à une instruction de contrôle se repère au : en fin d'instruction de contrôle et est délimité uniquement par l'indentation de 4 espaces relativement au début de l'instruction de contrôle.**
**Il est crucial, lors de la saisie du texte du programme de respecter les indentations du texte.**

---

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

---

```
a=0.1 #fin amont1
if a <= a*a : #debut bloc1
    print(a," est inferieur a son carre")
    print("coucou") #fin bloc1
print "on continue le programme"
a=1.1 #debut aval1, fin amont2
if a <= a*a : #debut bloc2
    print(a," est inferieur a son carre")
    print("coucou") #fin bloc2
print("c'est la derniere ligne") #debut aval2
```

---

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

---

Bien observer la façon dont la flèche verte se déplace par sauts (ou pas). On voit dans cet exemple deux blocs de programmes, marqués par les commentaires commençant par `#`, associés aux deux `if` successifs. Dans les deux cas l'objet booléen évalué est l'expression `a <= a*a`, issue du résultat d'une comparaison.

Une version plus élaborée de l'instruction `if` permet d'effectuer un bloc de programme si l'objet booléen testé est vrai et un autre bloc de programme s'il est faux : c'est la forme `if/else`.

Son insertion dans le programme suit le schéma suivant

```
amont
if p :
    blocV
else :
    blocF
aval
```

Le déroulement du programme est le suivant : la suite d'instructions `amont` est exécutée jusqu'à ce que le curseur arrive à `if p :`. Deux possibilités s'offrent alors

- Si `p` vaut `True`, le bloc de programme `blocV` est exécuté puis on saute directement à l'exécution de la suite d'instructions `aval`.
- Si `p` vaut `False`, le bloc de programme `blocV` n'est pas exécuté, le bloc de programme `blocF` est exécuté et on continue l'exécution de la suite d'instructions `aval`.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
a=0.1 #fin amont1
if a <= a*a : #debut blocV1
    print(a," est inferieur a son carre") #fin blocV1
else : #debut blocF1
    print(a," est superieur strict. a son carre") #fin blocF1
print("on continue le programme")
a=1.1 #debut aval1, fin amont2
if a <= a*a : #debut blocV2
    print(a," est inferieur a son carre") #fin blocV2
else : #debut blocF2
    print(a," est superieur strict. a son carre") #fin blocF2
print("c'est la derniere ligne") #debut aval2
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Un bloc de programme peut bien évidemment contenir des instructions de contrôle. En un sens, on peut imbriquer ces blocs de programme les uns dans les autres.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
a=0.1
if a*a <= 2*a :
    print("le carre de ",a," est inferieur a son double")
    if a*a<= a :
        print(a," est inferieur a son carre")
    else :
        print(a," est superieur strict. a son carre")
else :
    print("le carre de ",a," est strict. superieur a son double")
    if a*a<= a :
        print(a," est inferieur a son carre")
    else :
        print(a," est superieur strict. a son carre")
print("on continue le programme")
a=1.1
if a*a <= 2*a :
    print("le carre de ",a," est inferieur a son double")
    if a*a<= a :
        print(a," est inferieur a son carre")
    else :
        print(a," est superieur strict. a son carre")
else :
    print("le carre de ",a," est strict. superieur a son double")
    if a*a<= a :
        print(a," est inferieur a son carre")
    else :
        print(a," est superieur strict. a son carre")
print("on continue le programme")
a=-1.1
if a*a <= 2*a :
    print("le carre de ",a," est inferieur a son double")
    if a*a<= a :
        print(a," est inferieur a son carre")
    else :
        print(a," est superieur strict. a son carre")
else :
    print("le carre de ",a," est strict. superieur a son double")
    if a*a<= a :
        print(a," est inferieur a son carre")
    else :
        print(a," est superieur strict. a son carre")
print("on continue le programme")
a=2.1
if a*a <= 2*a :
    print("le carre de ",a," est inferieur a son double")
```

```
    if a*a<= a :
        print(a," est inferieur a son carre")
    else :
        print(a," est superieur strict. a son carre")
else :
    print("le carre de ",a," est strict. superieur a son double")
    if a*a<= a :
        print(a," est inferieur a son carre")
    else :
        print(a," est superieur strict. a son carre")
print("fin du programme")
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

On a la désagréable impression d'avoir recopié 4 fois la même chose, on réglera ce problème en 2.4.

Ceci dit, on a, en comptant grossièrement, 4 possibilités de sortie pour une valeur de $a$ donnée. Est-ce qu'une analyse mathématique un peu plus fine permettrait de montrer qu'un cas ne peut jamais arriver ?

Pour finir sur `if`, on peut signaler la construction `elif` permettant de contracter des `else if` successifs.

```
amont
if p1 :
    blocV1
elif p2 :
    blocF1F2V2
elif p3 :
    blocF1F2V3
...
else :
    blocF1F2F3
aval
```

### 2.3.2 Boucle avec condition de continuation : `while`

L'instruction `while` permet de recommencer une suite d'instructions–le *bloc de programme* associé à cette instruction de contrôle– **tant** qu'un certain objet booléen `p` est à la valeur `True`.

Son insertion dans le programme suit le schéma suivant

```
amont
while p :
    bloc
aval
```

Le déroulement du programme est le suivant : la suite d'instructions `amont` est exécutée jusqu'à ce que le curseur arrive à `while p :`. `p` est un objet (peut-être issu du calcul d'une expression) de type booléen. Deux possibilités s'offrent alors

- Si `p` vaut `True`, la suite d'instructions `bloc` est exécutée et l'on revient au début de `bloc`.
- Si `p` vaut `False`, la suite d'instructions `bloc` n'est pas exécutée et on saute directement à l'exécution de la suite d'instructions `aval`.

Il doit être clair que si on ne veut pas se trouver dans une boucle infinie, l'objet `p`, valant `True` si on entre dans cette boucle, doit être modifié par `bloc` pour permettre la sortie de la boucle.

Les codes suivants fournissent des boucles infinies, la dernière instruction n'est théoriquement jamais atteinte. Le deuxième code risque de mener à une interruption du programme (les entiers Python sont en nombre fini...)

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
while True : #debut bloc
   print("l'eternite est longue") #fin bloc
print("surtout vers la fin")
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
i=0
while i>=0 : #debut bloc
   print("l'eternite est longue")
   i=i+1 #fin bloc
print("surtout vers la fin")
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Une utilisation simple de la boucle `while` est la répétition d'une action un certain nombre de fois fixé à l'avance.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
i=0
nb_tours=10
while i<nb_tours :
   i=i+1
   print("c'est le tour numero", i)
print("On a fait ",i,"tours")
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

De même, l'exemple suivant trouve l'indice de la première occurrence d'un caractère dans une chaîne de caractères

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
s="abcdefgh"
car='e'
nb_tours_max=len(s)
car_trouve=False
i=0
while (i<nb_tours_max) and not(car_trouve) :
   car_trouve=(s[i]==car)
   i=i+1
if (i==nb_tours_max) :
   print(car,"n'est pas dans",s)
else
   print("La premiere occurence de",car,"dans",s,"est a l'indice",i-1)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

### 2.3.3 Itération : `for`

L'instruction `for` permet de parcourir, dans l'ordre, une liste (au sens large, ce peut être un t-uple, une chaîne de caractère,...) d'objets et d'appliquer une suite d'instructions–le *bloc de programme* associé à l'instruction `for`– à **chacun** des objets de la liste.

Son insertion dans le programme suit le schéma suivant

```
amont
for x in liste :
    bloc
aval
```

Le déroulement du programme est le suivant : la suite d'instructions `amont` est exécutée jusqu'à ce que le curseur arrive à `for x in liste :`. `liste` est une liste d'objets (ou un t-uple).

- l'étiquette `x` référence le premier objet de la liste et le bloc de programme `bloc` est exécuté,
- puis l'étiquette `x` référence le deuxième objet de la liste et le bloc de programme `bloc` est exécuté,
- puis l'étiquette `x` référence le troisième objet de la liste et le bloc de programme `bloc` est exécuté,
- ... et ainsi de suite jusqu'à épuisement de la liste.

la suite d'instructions `aval` est alors exécutée.

Une utilisation simple de la boucle `for` est la répétition d'une action un certain nombre de fois fixé à l'avance $N$. Il faut créer une liste des nombres entiers compris entre 0 et $N-1$ et « boucler »sur cette liste d'entiers. Observer l'utilisation de la fonction `range()` à cet effet.

En Python3.3, l'objet défini par `range(...)` **n'est pas** une liste d'entiers. Il s'agit d'un « itérateur », notion hors programme.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
nb_tours=10
for i in range(nb_tours) :
   print("c'est le tour numero", i)
print("On a fait ",i+1,"tours")
print(range(nb_tours))
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

On peut boucler sur d'autres listes et si les commandes utilisées le permettent, on peut boucler sur des listes d'objets hétérogènes.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
pi=3.14
e=2.71
ln2=0.69
liste=[e,pi,ln2]
for x in liste :
    print(x,"*",x,"="x**x)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
pi=3.14
e=2.71
ln2=0.69
txt1="abcdefgh"
txt2="uvwxyz"
liste=[e,txt1,pi,txt2,ln2]
for x in liste :
    print(x)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

et l'on peut bien sûr imbriquer des boucles, notamment lorsque l'on a des listes de listes–noter la définition de la liste qui s'étend sur plusieurs lignes

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
liste=[
       [0,1,2,3],
       ["truc","bidule","chouette"],
       ["truc", 0, 1j]
      ]
for x in liste :
    for y in x :
        print(y)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

## 2.4 Les fonctions, des objets presque comme les autres

Une fonction python est une manière de placer un bloc de programme dans un objet de sorte à pouvoir le réutiliser plusieurs fois, avec la possibilité de fixer les valeurs de certains paramètres.

Le comportement est en ceci similaire à celui d'une fonction mathématique. Lorsque l'on utilise la fonction $\exp$, on dispose d'une machine dont on ne connaît pas forcément le fonctionnement interne et qui étant donné un paramètre réel $x$, passé en *argument*, calcule ou *retourne* la valeur du nombre $\exp(x)$.

La définition d'une fonction se fait grâce à la directive `def` suivie d'une étiquette pour cette fonction (le nom de la fonction) ainsi que la liste des paramètres qui est une liste d'étiquettes placée entre parenthèses. Vient ensuite, dans un bloc de programme, la suite des instructions permettant le calcul de la valeur de retour de la fonction, valeur de retour qui sera renvoyée au moment où l'interpréteur rencontre l'instruction `return`.

Il peut ne pas y avoir de valeur de retour, par exemple dans le cas où une fonction s'occupe seulement de l'affichage. Le retour se fait alors à la fin du bloc de programme associé à la fonction.

**Une erreur commune et curieuse chez les débutants est la confusion entre `print` et `return` !**

- `print` « imprime »son argument à l'écran et la valeur de celui-ci est perdue,
- `return` passe la valeur en argument à l'endroit appelant la fonction, cete valeur peut-être récupérée (dans un identificateur) et utilisée à d'autres fins.

Voici par exemple la définition d'une fonction, étiquetée `f`, dont l'argument est un objet réel et qui retourne un nombre réel.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
def f(x):
    if x<0 :
        return 2.0*x
```

```
    else :
        return 3.0*x
a=f(2)
b=f(-4)
print(type(a),type(f))
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

— Le bloc de programme définissant `f` commence à la ligne 2 et finit à la ligne 5.
— Le bloc principal de programme comprend la définition de la fonction `f` et continue à la ligne 6.
— Observer particulièrement comment le pointeur de ligne se déplace lors des appels à la fonction, lignes 6 et 7.
— On voit que `a` est un objet de type nombre réel alors que `f` est un objet de type fonction.

Il est important de comprendre la notion de *portée* ou de *visibilité* d'une variable. On modifie un peu l'exemple précédent.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
def f(x):
    print(x)
    if x<0 :
        return 2.0*x
    else :
        return 3.0*x
x=3.0
print(x)
a=f(2)
print(x)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

On a, à dessein, utilisé la même étiquette `x` comme étiquette dans le bloc principal et comme étiquette dans le bloc de définition de la fonction. L'objet référencé par `x` dépend de l'endroit où l'on se trouve dans le texte du programme. En un sens, le fait d'utiliser `x` dans le bloc de définition de `f` « masque », dans ce bloc seulement, l'objet référencé par `x` dans le bloc principal du programme.

Dans la variante suivante, l'objet référencé par l'étiquette `y` utilisée dans le bloc principal est visible depuis le bloc de définition de la fonction. L'objet référencé par l'étiquette `x` utilisée dans le bloc principal n'est visible pas depuis le bloc de définition de la fonction.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
def f(x):
    print(x)
    if x<0 :
        return 2.0*x
    else :
        return y*x
x=3.0
print(x)
y=3
a=f(2)
print(a,y)
y=4
a=f(2)
print(a,y)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

L'utilisation de fonctions permet la *factorisation du code*, c'est à dire éviter le présence de bouts de code similaires à différents emplacements du code. On reprend l'exemple de la section sur `if`.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
def f(a):
    if a*a <= 2*a :
        print("le carre de ",a," est inferieur a son double")
        if a*a<= a :
            print(a," est inferieur a son carre")
        else :
            print(a," est superieur strict. a son carre")
    else :
        print("le carre de ",a," est strict. superieur a son double")
        if a*a<= a :
```

```
            print(a," est inferieur a son carre")
        else :
            print(a," est superieur strict. a son carre")

#debut du programme principal
f(0.1)
print("on continue le programme")
f(1.1)
print("on continue le programme")
f(-1.1)
print("on continue le programme")
f(2.1)
print("fin du programme")
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

On voit qu'on y a gagné au plan de la lisibilité et de la maintenance (si on doit changer un texte, il suffit de le changer une fois dans le corps de la fonction `f`). Le texte qui était écrit 4 fois ne l'est plus qu'une.

Cependant, ce code n'est pas entièrement factorisé. Voici une version améliorée.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
def g(a):
    if a*a<= a :
        print(a," est inferieur a son carre")
    else :
        print(a," est superieur strict. a son carre")

def f(a):
    if a*a <= 2*a :
        print("le carre de ",a," est inferieur a son double")
        g(a)
    else :
        print("le carre de ",a," est strict. superieur a son double")
        g(a)

def cont(): #une fonction sans argument
    print("on continue le programme")

#debut du programme principal
f(0.1)
cont()
f(1.1)
cont()
f(-1.1)
cont()
f(2.1)
print("fin du programme")
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Et finalement, la version la plus élégante, en utilisant une liste et une boucle `for`

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
def g(a):
    if a*a<= a :
        print(a," est inferieur a son carre")
    else :
        print(a," est superieur strict. a son carre")

def f(a):
    if a*a <= 2*a :
        print("le carre de ",a," est inferieur a son double")
    else :
        print("le carre de ",a," est strict. superieur a son double")

def cont(): #une fonction sans argument
    print("on continue le programme")
```

```
#debut du programme principal
liste=(0.1,1.1,-1.1,2.1)
imprime_cont=False
for x in liste :
    if(imprime_cont) :
        cont()
    f(x)
    g(x)
print("fin du programme")
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Pour finir, les objets fonctions peuvent être mis en listes, recevoir d'autres étiquettes que celle reçue initialement, être eux-mêmes arguments de fonctions.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
def f1(x):
    print(x)
    if x<0 :
        return 2.0*x
    else :
        return 3.0*x
def f2(x):
    return x
def f3(z):
    return z*z
def g(f,x):
    return x*f(x)
def h(f):
    def phi(x):
        return x*f(x)
    return phi
f=(f1,f2,f3)
print(type(f))
print((f[0])(2), f[1](2), f[2](2))
print(g(f[0],2))
psi=h(f[0])
print(type(psi))
print(psi(2))
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Pourquoi les résultats affichés aux lignes 20 et 23 sont-ils identiques ? Comprendre la nature des objets référencés par les étiquettes.

## 2.5 Modules externes

Lorsque l'on prend Python « à nu », seules les instructions de contrôle et objets et fonctions de base (comme `print`) sont définies–en anglais *built-in*– Il est clair que l'on ne peut réinventer la roue à chaque fois et des communautés de développeurs ont écrit et écrivent encore de nouveaux types d'objets et de nouvelles fonctions nous permettant d'avancer un peu. Ces nouvelles fonctionnalités sont regroupées dans des scripts particuliers, les *modules*.

On peut par exemple faire du calcul scientifique à la MatLab en utilisant les modules `numpy`, `scipy` et `sympy`, avoir des fonctionnalités graphiques avancées en utilisant le module `matplotlib`, etc...

PythonTutor n'admet le chargement que de très peu de modules.

A titre d'exemple, les fonctions mathématiques avancées (celles d'une calculatrice scientifique du commerce) sont regroupées dans un module standard nommé `math`. Nous ne nous en servirons pas pendant l'année mais PythonTutor accepte de le charger...

Nous commençons par « importer »le module dans notre script, ce qui nous permettra d'utiliser la fonction exponentielle dont l'étiquette originale est `math.exp`. De même, la constante réelle $\pi$ est définie dans module comme étant `math.pi`.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
import math
e=math.exp(1)
print(type(e),e)
print(type(math.exp))
print(math.pi)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

On peut surnommer le module avec la directive `as` afin d'alléger les écritures.

PythonTutor: Entrez le texte suivant dans la fenetre de code de PythonTutor,

```
import math as m
e=m.exp(1)
print(type(e),e)
print(type(m.exp))
print(m.pi)
```

Cliquez sur Visualize, avancez ou reculez avec Forward> et Backward< et observez la progression dans le code.

Pour aller plus loin et pouvoir utiliser nos modules de référence, nous allons nous placer dans un environnement de travail (Pyzo) nous laissant absolument toute liberté. Par ailleurs, du fait que c'est une bonne pratique de programmation en vue du projet, nous verrons comment écrire ses propres modules. Ceci permet de segmenter le travail à faire en sous-travaux ordonnés hiérarchiquement.

# 3 Utiliser Pyzo

Je suppose à présent que vous avez installé l'environnement Pyzo sur votre machine. Si vous avez des difficultés à le faire, demandez à une de vos connaissances de vous montrer comment faire. Les principales sources de connaissance en informatique étant, dans l'ordre

1. Internet et l'aide en ligne,
2. les amis,
3. le prof, si la situation est désespérée,

il ne faut pas hésiter à faire part de ses difficultés au monde entier.

Pyzo installe sur votre machine un interpréteur Python en version 3.3, des modules (`numpy`, `matplotlib`...), une console Python interactive (proche dans l'esprit de la console de Matlab pour ceux qui connaissent),...

Lancer Pyzo en cliquant sur la bonne icône à partir du dossier d'installation. On doit pouvoir installer cette icone dans le lanceur d'applications.

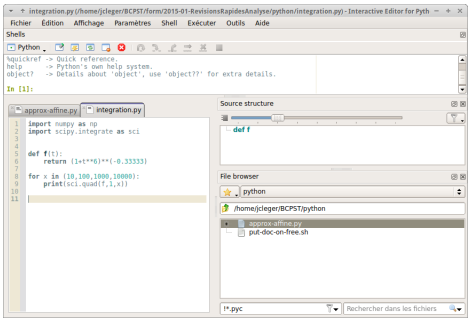

FIGURE 1 – Pyzo à l'ouverture

Par défaut, s'ouvre une fenêtre divisée en quatre parties, voir figure 1.

La partie en bas à gauche est l'éditeur de scripts. C'est là où l'on écrit les programmes destinés à être sauvés dans des fichiers.

La partie en haut est la console IPython :de là, on peut « lancer »un script, placer des commandes Python exécutées immédiatement, demander de l'aide (`help(objet)`, `help(commande)`), voir certains résultats graphiques, etc...

La partie en bas à droite comprend d'une part une description des objets définis dans le code source et d'autre part un explorateur de fichiers.

On peut changer ces fenêtres en allant dans le menu Outils.

## 3.1 Edition et execution de scripts

Lors des TD/TP ou de la réalisation de votre projet, vous aurez à écrire des scripts, que ce soit les programmes principaux ou des modules. La première chose à faire est d'organiser votre espace de travail en répertoires, sous-répertoires, etc...Mon conseil

est d'utiliser un répertoire par TD/TP et un répertoire pour le projet. Evitez, notamment pour le projet, de faire un répertoire par session de travail, la duplication des fichiers apporte beaucoup de confusion.

**Que ce soit pour les noms de repertoires, de fichiers, etc, n'utilisez que les caractères alphanumériques NON ACCENTUES et _, -, ne mettez pas d'espace ! ! !**

L'environnement de travail de Pyzo permet d'exécuter votre script, ceci peut se faire suivant deux modes :
— Le mode script : le répertoire de travail est alors celui de votre script, un nouvel interpréteur vide est lancé
— Le mode console : le script est excuté dans la console active.

Pour ce dernier mode, il faut faire attention au répertoire dans lequel on travaille, surtout à la première exécution. On peut afficher le nom du repertoire de travail par la commande `pwd`, lister son contenu par la commande `ls` et changer de repertoire par la commande `cd nouveau_rep`. La commande `cd ..` permet de remonter dans la hiérarchie des répertoires.

**Si vous n'êtes pas dans le bon repertoire de travail, Python ne trouvera pas vos données, les modules personnels, etc...**

L'édition d'un nouveau script se fait en cliquant sur le menu Fichier>Nouveau, la sauvegarde en cliquant sur Fichier>Sauvegarder, etc...

Pyzo: Ajoutez le texte suivant au fichier `hello.py`

```
print("bonjour le monde")
```

N'oubliez pas de sauver le fichier!

Pour exécuter le fichier courant, cliquer sur l'option désirée du menu Exécuter.
Le résultat du programme apparaît dans la console IPython.

Pyzo: Ajoutez le texte suivant au fichier `TeteAToto.py`

```
def f(i):
    print(i,"+",i)
    print("=")
    print(i+i)
    print("La tête à Toto")
f(0)
f(1)
```

N'oubliez pas de sauver le fichier!

Exécutez ! Les accents apparaissent-ils dans le texte écrit ?

S'ils n'apparaissent pas, ajouter dans la première ligne du script

```
# -*- coding: utf-8 -*-
```

D'une manière générale, il est préférable d'avoir en entête de chaque fichier qq lignes de code du type

```
# -*- coding: utf-8 -*-
"""
Created on Date à remplir

@author: votre nom
"""
```

ce qui identifie le fichier et élimine les problèmes de caractères spéciaux.

## 3.2 Le mode interactif : IPython

On peut rentrer directement des commandes Python dans la console IPython, cela peut-être utile pour tester le résultat de scripts, faire une représentation graphique rapide, etc...

Cette console s'utilise un peu comme une calculatrice ou une application de type Maple ou Matlab. Par exemple, supposons que l'on travaille avec une fonction définie dans un script.

Pyzo: Ajoutez le texte suivant au fichier `xexpx.py`

```
import numpy as np
import matplotlib.pyplot as plt
def f(x):
    return x*np.exp(x)
```

N'oubliez pas de sauver le fichier!

Exécutez ce script. La fonction `f` est maintenant définie, *i.e.* il s'agit d'un objet Python occupant une place physique en mémoire, et nous pouvons l'utiliser dans la console. Si par exemple, nous voulons en tracer le graphe,

IPython: Entrez les lignes suivantes dans la console IPython

```
x=np.linspace(0.0,0.5,500)
plt.plot(x,f(x))
```

## 3.3 Utiliser les modules

Dans le script `xexpx.py` de l'exemple précédent, il y a deux commandes d'utilisation de modules (`import`).

Les deux modules concernés, `numpy` et `mathplotlib.pyplot` sont des modules dédiés au calcul scientifique. Le code les renomme respectivement `np` et `plt`, ce qui est la façon standard pour les surnommer.

Les sites `http://matplotlib.org/` et `http://www.numpy.org/` contiennent, outre la documentation complète de ces modules (en anglais), des exemples d'utilisation très très instructifs.

Nous les utiliserons systématiquement. Ceci signifie que vos scripts contiendront quasiment toujours ces deux directives d'importation

La fonction `np.exp` est la fonction `exp` définie dans le module surnommé `np`, i.e. le module `numpy`.

La fonction `plt.plot` est la fonction `plot` définie dans le module surnommé `plt`,i.e. le (sous)-module `matplotlib.pyplot`

On peut écrire ses propres modules sans aucune difficulté, ce qui permet de placer les fonctions relatives à un type d'objet personnalisé dans un fichier autonome et donc de segmenter le travail.

Pyzo: Ajoutez le texte suivant au fichier `monmodule.py`

```
def mafonction(x):
    print("Ma première fonction en module, son argument vaut:",x)
```

N'oubliez pas de sauver le fichier!

Pyzo: Ajoutez le texte suivant au fichier `testmodule.py`

```
import monmodule as mm
mm.mafonction(2)
mm.mafonction("haha")
```

N'oubliez pas de sauver le fichier!

Exécutez uniquement `testmodule.py`. Il donne le résultat escompté si le module est dans le même répertoire que ce script.

On peut avoir des scripts qui sont à la fois des modules et des programmes principaux. Cela permet, dans le corps du script d'inclure des tests des fonctionnalités du module. Ces tests ne sont pas exécutés lorsque le script est utilisé en tant que module.

Pyzo: Modifiez le fichier `monmodule.py` de la façon suivante

```
def mafonction(x):
    print "Ma première fonction en module, son argument vaut:",x

#au cas ou ce script serait appele directement, on peut mettre des
#tests du module
if __name__=='__main__':
    mafonction(1)
    mafonction("truc")
```

N'oubliez pas de sauver le fichier!

## 3.4 Système d'aide, documenter et commenter son code

Pour obtenir la syntaxe d'une fonction particulière ou la documentation d'un objet, il suffit d'utiliser la fonction `help()` dans la console. Par exemple, en tapant `help(np.exp)` dans la console IPython, on obtient la documentation de la fonction `exp` du module `np`, *a.k.a* `numpy`.

Faites maintenant la même chose en tapant `help(f)`, où `f` est l'étiquette de la fonction `f` que l'on définit dans le script. Le résultat n'est que peu signifiant....

On aimerait bien (surtout si on doit réutiliser les objets définis) qu'apparaisse un mode d'emploi succinct des objets que nous définissons.

Pyzo: Modifiez le fichier `xexpx.py` de la façon suivante

```
import numpy as np
import matplotlib.pyplot as plt
def f(x):
    """
    Calcule x.exp(x)
    """
    return x*np.exp(x) #on utilise np.exp qui applique l'exponentielle
                       #a un tableau de nombres
```

N'oubliez pas de sauver le fichier!

Executez et utilisez l'inspecteur sur `f`. C'est plus satisfaisant ?
Les documentations de niveau professionnel ressemblent à ceci

Pyzo: Modifiez le fichier `xexpx.py` de la façon suivante

```
import numpy as np
import matplotlib.pyplot as plt
def f(x):
    """
    Calcule x.exp(x)

    Parameters
    ----------
    x : ndarray
        Un tableau de nombres reels

    Returns
    -------
    res: ndarray
         La fonction x->x.exp(x) appliquee a x
    """
    return x*np.exp(x)
```

N'oubliez pas de sauver le fichier!

Executez et utilisez l'inspecteur sur `f`. C'est encore plus satisfaisant.
Le code que vous écrivez doit être **intégralement** documenté et commenté.
Un commentaire commence par le caractère `#`. Les commentaires ne sont pas lus par l'interpréteur et sont destinés à vous même ou à toute autre personne qui doit lire et comprendre votre code.

**Il est extrêmement important de documenter votre travail et d'y placer des commentaires ! ! !**